## **TensorFlow Frame**

## **Guide Pamphlet**

# tim

**The abbreviations of tensorflow in this pamphlet are all ‘tf’ or ‘Tf’.**

**FUNDAMENTAL KNOWLEGE**

**1、Placeholder of data input:**

**Function**:

tf.placeholder(dtype , shape,name)

**Example**:

Input = tf.placeholder(dtype = tf.float32 , shape = [None , 17,19,19])

**Annotation**:

None means the dimension of batch size . In the placeholder function , it can be set with None key word.

**2、Variable establish :**

**Function**(omit some parameters which are not important in this function):

tf.Variable(initial\_value , trainable)

**Example**:

tf.Variable(initial\_value = tf.truncated\_normal(shape = [256 , 19 \* 19 + 1],dtype = tf.float32),trainable = True)

**Annotation**:

trainable is an very very essential parameter , in the GAN net , we must control the variables which can be trained and which can not be trained.

**3、add\_to\_collection()**

**Example**:

weights = tf.Variable()

tf.add\_to\_collection(tf.GraphKeys.WEIGHTS, weights)

**Annotation**:

In the tensorflow , there is an Graph in it and it contains prodigious variables and optional.So, this function is helping the variable to participate an collection which will help you to classify your variables.

The list of collections which are used with higher frequent.

tf.GraphKeys.WEIGHTS (members in this collection are not added automatic , so we need to add weight to this collection with a manual operation)

tf.GraphKeys.GLOBAL\_VARIABLES (auto)

tf.GraphKeys.TRAINABLE\_VARIABLES (auto)

tf.GraphKeys.UPDATE\_OPS (auto)

**4、get\_collection()**

**Example**:

tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

**Annotation**:

Return an list of variables in the collection which would you want to get .

1. **identity()**

**Example:**

orig = tf.identity(net)

**Annotation:**

Deep duplicate the Tensor in Tensorflow .

1. **clip\_by\_value(t，clip\_value\_min ,clip\_value\_max, name) :**

t – A `Tensor`.

clip\_value\_min – A 0-D (scalar) `Tensor`, or a `Tensor` with the same shape as `t`. The minimum value to clip by.

clip\_value\_max – A 0-D (scalar) `Tensor`, or a `Tensor` with the same shape as `t`. The maximum value to clip by.

name – A name for the operation (optional).

**Example:**

tf.clip\_by\_value(y,clip\_value\_min=0.2,clip\_value\_max=1)

**Annotation:**

Given a tensor t, this operation returns a tensor of the same type and shape as t with its values clipped to clip\_value\_min and clip\_value\_max. Any values less than clip\_value\_min are set to clip\_value\_min. Any values greater than clip\_value\_max are set to clip\_value\_max.

1. **exponential\_decay(learning\_rate, global\_step, decay\_steps, decay\_rate, staircase,name)**

learning\_rate – A scalar `float32` or `float64` `Tensor` or a Python number. The initial learning rate.

global\_step – A scalar `int32` or `int64` `Tensor` or a Python number. Global step to use for the decay computation. Must not be negative.

decay\_steps – A scalar `int32` or `int64` `Tensor` or a Python number. Must be positive. See the decay computation above.

decay\_rate – A scalar `float32` or `float64` `Tensor` or a Python number. The decay rate.

staircase – Boolean. If `True` decay the learning rate at discrete intervals

name – String. Optional name of the operation. Defaults to 'ExponentialDecay'.

**Example:(decay every 100000 steps with a base of 0.96)**

global\_step = tf.Variable(0, trainable=False)

starter\_learning\_rate = 0.1

learning\_rate = tf.train.exponential\_decay(starter\_learning\_rate, global\_step,

100000, 0.96, staircase=True)

# Passing global\_step to minimize() will increment it at each step.

tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss..., global\_step=global\_step)

**Annotation:**

Applies exponential decay to the learning rate.

When training a model, it is often recommended to lower the learning rate as the training progresses. This function applies an exponential decay function to a provided initial learning rate. It requires a global\_step value to compute the decayed learning rate. You can just pass a TensorFlow variable that you increment at each training step.

The function returns the decayed learning rate. It is computed as:

```python

decayed\_learning\_rate = learning\_rate \*decay\_rate ^ (global\_step / decay\_steps)

```

If the argument staircase is True, then global\_step / decay\_steps is an integer division and the decayed learning rate follows a staircase function.

**ACTIVATION FUNCTIONS**

**Activation functions that we will be used in our daily life.**

1、tf.nn.sigmoid(x)

2、tf.nn.tanh(x)

3、tf.nn.relu(x)

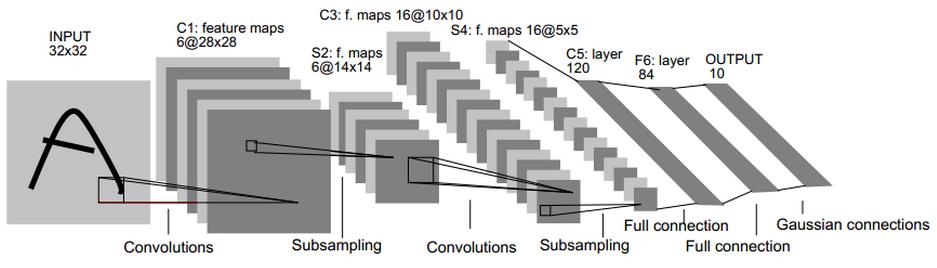
4、tf.nn.elu(x)

5、tf.nn.selu(x)

**CONVOLUTION FUNCTIONS**

**Additional**:

As usual , the size of convolution net inputs is [batch\_size , channels , image\_height , image\_width ]



**1、conv2d()**

Computes a 2-D convolution given 4-D input and filter tensors.

**Function**:

tf.nn.conv2d(input,filter,strides,padding,data\_format="NHWC")

input – A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. A 4-D tensor. The dimension order is interpreted according to the value of `data\_format`, see below for details.

filter – A `Tensor`. Must have the same type as `input`. A 4-D tensor of shape `[filter\_height, filter\_width, in\_channels, out\_channels]`

strides – A list of `ints`. 1-D tensor of length 4. The stride of the sliding window for each dimension of `input`. The dimension order is determined by the value of `data\_format`, see below for details.

padding – A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.

data\_format – An optional `string` from: `"NHWC", "NCHW"`. Defaults to `"NHWC"`. Specify the data format of the input and output data. With the default format "NHWC", the data is stored in the order of: [batch, height, width, channels]. Alternatively, the format could be "NCHW", the data storage order of: [batch, channels, height, width].

**Example**：

Input = tf.placeholder(dtype = tf.float32 , shape = [None , 17,19,19])

Filters = tf.Variable(tf..truncated\_normal(shape = [2,2,17,256],dtype = tf.float32))

Conv1 = tf.nn.conv2d(input = input , filter = Filters , strides = [1,1,1,1] , padding = “SAME”,data\_format = “NCHW”)

**Annotation**:

Given an

input tensor of shape [batch, in\_height, in\_width, in\_channels]

And a filter/kernel tensor of shape[filter\_height, filter\_width, in\_channels, out\_channels],

In detail, with the default NHWC format,

output[b, i, j, k] =

sum\_{di, dj, q} input[b, strides[1] \* i + di, strides[2] \* j + dj, q] \*

filter[di, dj, q, k]

Must have strides[0] = strides[3] = 1. For the most common case of the same horizontal and vertices strides, strides = [1, stride, stride, 1].

**2、batch\_normalization()**

**Function**:

tf.layers.batch\_normalization(inputs,axis=-1,momentum=0.99,epsilon=1e-3,fused=None,center=True,scale=True,training=False,trainable=True)

inputs – Tensor input.

axis – An `int`, the axis that should be normalized (typically the features axis). For instance, after a `Convolution2D` layer with `data\_format="channels\_first"`, set `axis=1` in `BatchNormalization`.

momentum – Momentum for the moving average.

epsilon – Small float added to variance to avoid dividing by zero.

center – If True, add offset of `beta` to normalized tensor. If False, `beta` is ignored.

scale – If True, multiply by `gamma`. If False, `gamma` is not used. When the next layer is linear (also e.g. `nn.relu`), this can be disabled since the scaling can be done by the next layer.

fused – if `None` or `True`, use a faster, fused implementation if possible. If `False`, use the system recommended implementation.

trainable – Boolean, if `True` also add variables to the graph collection `GraphKeys.TRAINABLE\_VARIABLES` (see tf.Variable).

training – Either a Python boolean, or a TensorFlow boolean scalar tensor (e.g. a placeholder). Whether to return the output in training mode (normalized with statistics of the current batch) or in inference mode (normalized with moving statistics). \*\*NOTE\*\*: make sure to set this parameter correctly, or else your training/inference will not work properly.

**Example**:

net = tf.layers.batch\_normalization(net,

epsilon=1e-5, axis=1, fused=True,

center=True, scale=False,

training=self.training)

**Annotation**:

"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift"

Note: when training, the moving\_mean and moving\_variance need to be updated. By default the update ops are placed in tf.GraphKeys.UPDATE\_OPS, so they need to be added as a dependency to the train\_op. Also, be sure to add any batch\_normalization ops before getting the update\_ops collection. Otherwise, update\_ops will be empty, and training/inference will not work properly. For example:

```python

x\_norm = tf.layers.batch\_normalization(x, training=training)

# ...

update\_ops = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

with tf.control\_dependencies(update\_ops):

train\_op = optimizer.minimize(loss)

```

The BN layer always adds before the activation layer.

If the parameter training is set False , it would not add anything to the collection ,

but , if the training is set TRUE , it would participate operation to the UPDATE \_OPS collection.Because it has beta and gamma parameters which need machine to learn by itself .

**3、pool()**

**Function:**

Tf.nn.pool(input,window\_shape,pooling\_type,padding,data\_format=None)

input – Tensor of rank N+2, of shape `[batch\_size] + input\_spatial\_shape + [num\_channels]` if data\_format does not start with "NC" (default), or `[batch\_size, num\_channels] + input\_spatial\_shape` if data\_format starts with "NC". Pooling happens over the spatial dimensions only.

window\_shape – Sequence of N ints >= 1

pooling\_type – Specifies pooling operation, must be "AVG" or "MAX".

padding – The padding algorithm, must be "SAME" or "VALID"

data\_format - it is same with conv2d() function. And it must same with the data\_format in the function conv2d() . if it it not same , the program will report error.

**Example:**

result=tf.nn.conv2d(x,W,strides=[1,1,1,1],padding='SAME',data\_format="NCHW")

poolR=tf.nn.pool(input=result,pooling\_type="MAX",window\_shape=[2,2],padding="SAME",data\_format="NCHW")

**4、conv2d\_transpose()**

**Function:**

Tf.nn.conv2d\_transpose( value, filter, output\_shape,strides, padding="SAME", data\_format="NHWC", name=None)

value – A 4-D `Tensor` of type `float` and shape `[batch, height, width, in\_channels]` for `NHWC` data format or `[batch, in\_channels, height, width]` for `NCHW` data format.

filter – A 4-D `Tensor` with the same type as `value` and shape `[height, width, output\_channels, in\_channels]`. `filter`'s `in\_channels` dimension must match that of `value`.

output\_shape – A 1-D `Tensor` representing the output shape of the deconvolution op.

strides – A list of ints. The stride of the sliding window for each dimension of the input tensor.

padding – A string, either `'VALID'` or `'SAME'`. The padding algorithm.

data\_format – A string. 'NHWC' and 'NCHW' are supported.

name – Optional name for the returned tensor.

**Example:**

inputD=tf.Variable(initial\_value=tf.truncated\_normal(shape=[2,18,10,10],dtype=tf.float32),trainable=False)

filters=tf.Variable(initial\_value=tf.truncated\_normal(shape=[3,3,1,18],dtype=tf.float32),trainable=True)

image = tf.nn.conv2d\_transpose(value=inputD,filter=filters,

output\_shape=[1,1,19,19],strides=[1,1],

padding = "VALID",data\_format="NCHW")

**RNN FUNCTIONS**

**Additional :**

As usual , the size of RNN or LSTM input net is [batch\_size , time\_steps , word\_vector]

**1、Basic cell unit construction**

**Functions:**

tf.nn.rnn\_cell.BasicLSTMCell(num\_units)

tf.nn.rnn\_cell.BasicRNNCell(num\_units)

Num\_units -- the number of units you want to have in a single cell .

**Example:**

tf.nn.rnn\_cell.BasicLSTMCell(num\_units=10)

tf.nn.rnn\_cell.BasicRNNCell(num\_units=10)

1. **MultiRNNCell()**

**Function:**

tf.nn.rnn\_cell.MultiRNNCell(cells)

cells – list of RNNCells that will be composed in this order.

**Example:**

def LSTM\_Cell(num\_unites):

return tf.nn.rnn\_cell.BasicLSTMCell(num\_units=num\_unites)

la1 = LSTM\_Cell(30)

la2 = LSTM\_Cell(45)

la3 = LSTM\_Cell(50)

net1 = tf.nn.rnn\_cell.MultiRNNCell(cells = [la1,la2,la3])

**Annotation:**

The return of this function is a child class object of BasicLSTMCell or BasicRNNCell . it has all functions which belong to the classes that i have mentioned above .

**3、zero\_state()**

**Function:**

zero\_state(batch\_size, dtype)

batch\_size – int, float, or unit Tensor representing the batch size.

dtype – the data type to use for the state.

**Example:**

def LSTM\_Cell(num\_unites):

return tf.nn.rnn\_cell.BasicLSTMCell(num\_units=num\_unites)

la1 = LSTM\_Cell(30)

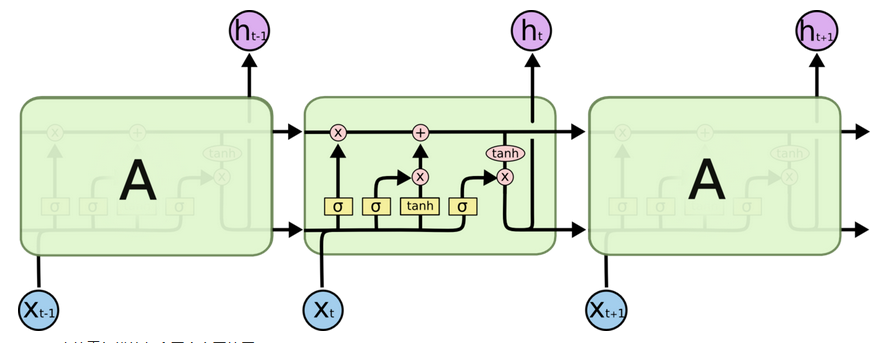
initialState = la1.zero\_state(batch\_size=1,dtype=tf.float32)

**Annotation:**

This function is used to build the initial state for the first time step in RNN or LSTM net work . As usual , its are all zero .

**4、dynamic\_rnn()**

**(Creates a recurrent neural network specified by RNNCell `cell`.Performs fully dynamic unrolling of `inputs`.)**

****

**Function:**

tf.nn.dynamic\_rnn(cell , inputs, initial\_state , dtype , time\_major=False)

cell: An instance of RNNCell.

inputs: The RNN inputs.

If `time\_major == False` (default), this must be a `Tensor` of shape:

`[batch\_size, max\_time, ...]`, or a nested tuple of such

elements.

If `time\_major == True`, this must be a `Tensor` of shape:

`[max\_time, batch\_size, ...]`, or a nested tuple of such

elements.

This may also be a (possibly nested) tuple of Tensors satisfying

this property. The first two dimensions must match across all the inputs,

but otherwise the ranks and other shape components may differ.

In this case, input to `cell` at each time-step will replicate the

structure of these tuples, except for the time dimension (from which the

time is taken).

The input to `cell` at each time step will be a `Tensor` or (possibly

nested) tuple of Tensors each with dimensions `[batch\_size, ...]`.

initial\_state: (optional) An initial state for the RNN.

If `cell.state\_size` is an integer, this must be

a `Tensor` of appropriate type and shape `[batch\_size, cell.state\_size]`.

If `cell.state\_size` is a tuple, this should be a tuple of

tensors having shapes `[batch\_size, s] for s in cell.state\_size`.

dtype: (optional) The data type for the initial state and expected output.

Required if initial\_state is not provided or RNN state has a heterogeneous

dtype.

time\_major: The shape format of the `inputs` and `outputs` Tensors.

If true, these `Tensors` must be shaped `[max\_time, batch\_size, depth]`.

If false, these `Tensors` must be shaped `[batch\_size, max\_time, depth]`.

Using `time\_major = True` is a bit more efficient because it avoids

transposes at the beginning and end of the RNN calculation. However,

most TensorFlow data is batch-major, so by default this function

accepts input and emits output in batch-major form.

**Example1:(one layer)**

```python

# create a BasicRNNCell

rnn\_cell = tf.nn.rnn\_cell.BasicRNNCell(hidden\_size)

# 'outputs' is a tensor of shape [batch\_size, max\_time, cell\_state\_size]

# defining initial state

initial\_state = rnn\_cell.zero\_state(batch\_size, dtype=tf.float32)

# 'state' is a tensor of shape [batch\_size, cell\_state\_size]

outputs, state = tf.nn.dynamic\_rnn(rnn\_cell, input\_data,

initial\_state=initial\_state,

dtype=tf.float32)

```

**Example2:(double layers)**

```python

# create 2 LSTMCells

rnn\_layers = [tf.nn.rnn\_cell.LSTMCell(size) for size in [128, 256]]

# create a RNN cell composed sequentially of a number of RNNCells

multi\_rnn\_cell = tf.nn.rnn\_cell.MultiRNNCell(rnn\_layers)

# 'outputs' is a tensor of shape [batch\_size, max\_time, 256]

# 'state' is a N-tuple where N is the number of LSTMCells containing a

# tf.contrib.rnn.LSTMStateTuple for each cell

outputs, state = tf.nn.dynamic\_rnn(cell=multi\_rnn\_cell,

inputs=data,

dtype=tf.float32)

**Example3:**

inputD = tf.placeholder(shape=[1,10,25],dtype=tf.float32)

def LSTM\_Cell(num\_unites):

return tf.nn.rnn\_cell.BasicLSTMCell(num\_units=num\_unites)

la1 = LSTM\_Cell(30)

la2 = LSTM\_Cell(45)

la3 = LSTM\_Cell(50)

net1 = tf.nn.rnn\_cell.MultiRNNCell(cells = [la1,la2,la3])

outputs , states = tf.nn.dynamic\_rnn(cell = net1 , inputs = inputD, dtype = tf.float32)

**Annotation:**

**Returns:**

A pair (outputs, state) where:

**outputs**: The RNN output `Tensor`.

If time\_major == False (default), this will be a `Tensor` shaped:

`[batch\_size, max\_time, cell.output\_size]`.

If time\_major == True, this will be a `Tensor` shaped:

`[max\_time, batch\_size, cell.output\_size]`.

Note, if `cell.output\_size` is a (possibly nested) tuple of integers

or `TensorShape` objects, then `outputs` will be a tuple having the

same structure as `cell.output\_size`, containing Tensors having shapes

corresponding to the shape data in `cell.output\_size`.

**state:** **The final state**. If `cell.state\_size` is an int, this

will be shaped `[batch\_size, cell.state\_size]`. If it is a

`TensorShape`, this will be shaped `[batch\_size] + cell.state\_size`.

If it is a (possibly nested) tuple of ints or `TensorShape`, this will

be a tuple having the corresponding shapes. If cells are `LSTMCells`

`state` will be a tuple containing a `LSTMStateTuple` for each cell.

**Example of Result:**

outputs is Tensor("rnn/transpose\_1:0", shape=(1, 10, 50), dtype=float32)

States is (LSTMStateTuple(c=<tf.Tensor 'rnn/while/Exit\_3:0' shape=(1, 30) dtype=float32>, h=<tf.Tensor 'rnn/while/Exit\_4:0' shape=(1, 30) dtype=float32>), LSTMStateTuple(c=<tf.Tensor 'rnn/while/Exit\_5:0' shape=(1, 45) dtype=float32>, h=<tf.Tensor 'rnn/while/Exit\_6:0' shape=(1, 45) dtype=float32>), LSTMStateTuple(c=<tf.Tensor 'rnn/while/Exit\_7:0' shape=(1, 50) dtype=float32>, h=<tf.Tensor 'rnn/while/Exit\_8:0' shape=(1, 50) dtype=float32>))

**5、bidirectional\_dynamic\_rnn()**

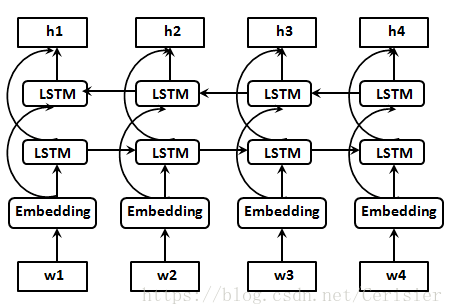
**(Creates a dynamic version of bidirectional recurrent neural network.**

**Takes input and builds independent forward and backward RNNs. The input\_size**

**of forward and backward cell must match. The initial state for both directions**

**is zero by default (but can be set optionally) and no intermediate states are**

**ever returned)**

****

**Function:**

Tf.nn.bidirectional\_dynamic\_rnn(cell\_fw,cell\_bw,inputs,initial\_state\_fw=None,initial\_state\_bw=None,dtype=None,time\_major=False）

cell\_fw: An instance of RNNCell, to be used for forward direction.

cell\_bw: An instance of RNNCell, to be used for backward direction.

inputs: The RNN inputs.

If time\_major == False (default), this must be a tensor of shape:

`[batch\_size, max\_time, ...]`, or a nested tuple of such elements.

If time\_major == True, this must be a tensor of shape:

`[max\_time, batch\_size, ...]`, or a nested tuple of such elements.

initial\_state\_fw: (optional) An initial state for the forward RNN.

This must be a tensor of appropriate type and shape

`[batch\_size, cell\_fw.state\_size]`.

If `cell\_fw.state\_size` is a tuple, this should be a tuple of

tensors having shapes `[batch\_size, s] for s in cell\_fw.state\_size`.

initial\_state\_bw: (optional) Same as for `initial\_state\_fw`, but using

the corresponding properties of `cell\_bw`.

dtype: (optional) The data type for the initial states and expected output.

Required if initial\_states are not provided or RNN states have a

heterogeneous dtype.

time\_major: The shape format of the `inputs` and `outputs` Tensors.

If true, these `Tensors` must be shaped `[max\_time, batch\_size, depth]`.

If false, these `Tensors` must be shaped `[batch\_size, max\_time, depth]`.

Using `time\_major = True` is a bit more efficient because it avoids

transposes at the beginning and end of the RNN calculation. However,

most TensorFlow data is batch-major, so by default this function

accepts input and emits output in batch-major form.

**Example:**

inputD = tf.placeholder(shape=[1,10,25],dtype=tf.float32)

def LSTM\_Cell(num\_unites):

return tf.nn.rnn\_cell.BasicLSTMCell(num\_units=num\_unites)

la1 = LSTM\_Cell(30)

la2 = LSTM\_Cell(45)

la3 = LSTM\_Cell(50)

F = tf.nn.rnn\_cell.MultiRNNCell(cells = [la1,la2,la3])

B = tf.nn.rnn\_cell.MultiRNNCell(cells=[la1,la2,la3])

outputsB , statesB = tf.nn.bidirectional\_dynamic\_rnn(F,B,inputs = inputD,dtype = tf.float32)

**Annotation:**

**Returns:**

A tuple (outputs, output\_states) where:

**outputs:** A tuple (output\_fw, output\_bw) containing the forward and the backward rnn output `Tensor`.

If time\_major == False (default),

output\_fw will be a `Tensor` shaped:

`[batch\_size, max\_time, cell\_fw.output\_size]`

and output\_bw will be a `Tensor` shaped:

`[batch\_size, max\_time, cell\_bw.output\_size]`.

If time\_major == True,

output\_fw will be a `Tensor` shaped:

`[max\_time, batch\_size, cell\_fw.output\_size]`

and output\_bw will be a `Tensor` shaped:

`[max\_time, batch\_size, cell\_bw.output\_size]`.

It returns a tuple instead of a single concatenated `Tensor`, unlike

in the `bidirectional\_rnn`. If the concatenated one is preferred,

the forward and backward outputs can be concatenated as

`tf.concat(outputs, 2)`.

**output\_states:** A tuple (output\_state\_fw, output\_state\_bw) containing

the forward and the backward **final states** of bidirectional rnn.

**Example of Result:**

outputB is (<tf.Tensor 'bidirectional\_rnn/fw/fw/transpose\_1:0' shape=(1, 10, 50) dtype=float32>, <tf.Tensor 'ReverseV2:0' shape=(1, 10, 50) dtype=float32>)

statesB is

((LSTMStateTuple(c=<tf.Tensor 'bidirectional\_rnn/fw/fw/while/Exit\_3:0' shape=(1, 30) dtype=float32>, h=<tf.Tensor 'bidirectional\_rnn/fw/fw/while/Exit\_4:0' shape=(1, 30) dtype=float32>),

LSTMStateTuple(c=<tf.Tensor 'bidirectional\_rnn/fw/fw/while/Exit\_5:0' shape=(1, 45) dtype=float32>, h=<tf.Tensor 'bidirectional\_rnn/fw/fw/while/Exit\_6:0' shape=(1, 45) dtype=float32>),

LSTMStateTuple(c=<tf.Tensor 'bidirectional\_rnn/fw/fw/while/Exit\_7:0' shape=(1, 50) dtype=float32>, h=<tf.Tensor 'bidirectional\_rnn/fw/fw/while/Exit\_8:0' shape=(1, 50) dtype=float32>)),

(LSTMStateTuple(c=<tf.Tensor 'bidirectional\_rnn/bw/bw/while/Exit\_3:0' shape=(1, 30) dtype=float32>, h=<tf.Tensor 'bidirectional\_rnn/bw/bw/while/Exit\_4:0' shape=(1, 30) dtype=float32>),

LSTMStateTuple(c=<tf.Tensor 'bidirectional\_rnn/bw/bw/while/Exit\_5:0' shape=(1, 45) dtype=float32>, h=<tf.Tensor 'bidirectional\_rnn/bw/bw/while/Exit\_6:0' shape=(1, 45) dtype=float32>),

LSTMStateTuple(c=<tf.Tensor 'bidirectional\_rnn/bw/bw/while/Exit\_7:0' shape=(1, 50) dtype=float32>, h=<tf.Tensor 'bidirectional\_rnn/bw/bw/while/Exit\_8:0' shape=(1, 50) dtype=float32>)))

**PREVENT OVERFIT**

1. **Dropout**

**Example1:**

y = tf.nn.dropout(x=x, keep\_prob=0.8)

x – A floating point tensor.

keep\_prob – A scalar `Tensor` with the same type as x. The probability that each element is kept.

**Annotation:**

The weights in x tensor will be amplified with an appropriate proportion which is 1 / keep\_prob if the weight will not be chosen to zero .

The return y has the same size of input x . the different between them is the values.

This function can be used in CNN or MLP network . it is the most ordinary dropout function in the tensorflow frame .

**Function2:**

tf.nn.rnn\_cell.DropoutWrapper(cell,input\_keep\_prob,output\_keep\_prob,state\_keep\_prob,dtype)

cell – an RNNCell, a projection to output\_size is added to it.

input\_keep\_prob – unit Tensor or float between 0 and 1, input keep probability; if it is constant and 1, no input dropout will be added.

output\_keep\_prob – unit Tensor or float between 0 and 1, output keep probability; if it is constant and 1, no output dropout will be added.

state\_keep\_prob – unit Tensor or float between 0 and 1, output keep probability; if it is constant and 1, no output dropout will be added. State dropout is performed on the outgoing states of the cell. \*\*Note\*\* the state components to which dropout is applied when `state\_keep\_prob` is in `(0, 1)` are also determined by the argument `dropout\_state\_filter\_visitor` (e.g. by default dropout is never applied to the `c` component of an `LSTMStateTuple`).

dtype – (optional) The `dtype` of the input, state, and output tensors.

**Example2:**

enc\_cell = tf.nn.rnn\_cell.MultiRNNCell(

[tf.nn.rnn\_cell.DropoutWrapper(tf.nn.rnn\_cell.BasicLSTMCell(rnn\_sizes), input\_keep\_prob=1.0, output\_keep\_prob=1.0) for \_ in range(num\_layers)])

**Annotation:**

Create a cell with added input, state, and/or output dropout.

Note, by default (unless a custom dropout\_state\_filter is provided), the memory state (c component of any LSTMStateTuple) passing through a DropoutWrapper is never modified. This behavior is described in the above article.

And this function can only be used in the LSTM or RNN cells , it can not be used in others variables like CNN or MLP.

1. **Regularizer**

**Example1:**

def weights\_With\_L2(lambda):

#create weights

we=tf.Variable(initial\_value=tf.truncated\_normal(shape=[2,19,19],dtype=tf.float32))

#calculate l2 regularizer

l2 = tf.nn.l2\_loss(we)

#add l2 \* lambda to loss collection

tf.add\_to\_collection("loss",tf.multiply(l2,lambda))

return we

w1 = weights\_With\_L2(0.01)

w2 = weights\_With\_L2(0.01)

loss = tf.reduce\_mean(tf.squared\_difference(w1,w2))

tf.add\_to\_collection("loss",loss)

#calculate gross loss with loss collection

tloss = tf.add\_n(tf.get\_collection("loss"))

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

print(sess.run(loss))

print(sess.run(tloss))

**Result:**

1.5487877

7.3386436

**Function:**

tf.nn.l2\_loss(t)

t – A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. Typically 2-D, but may have any dimensions.

**Annotation:**

L2 Loss.

Computes half the L2 norm of a tensor without the sqrt:

output = sum(t \*\* 2) / 2

**Example2:**

reg = tf.contrib.layers.l2\_regularizer(scale=0.0001)

regVar = tf.get\_collection(tf.GraphKeys.WEIGHTS)

regm = tf.contrib.layers.apply\_regularization(reg, regVar)

self.lossT = 1.0 \* lossP + 1.0 \* lossV + regm

**Annotation:**

Step 1 : establish an l2 regularizer object. Scale means the coefficient which will multiple with l2 regularizer .

Step 2 : get the weights in the collection of tf.GraphKeys.WEIGHTS

Step 3:apply regularizer to the weights

Step 4:add regularizer to gross loss.

**LOSS FORMULATION CONSTRUCTION**

**1、Loss function construction**

**Example1:**

cross = tf.nn.softmax\_cross\_entropy\_with\_logits(logits=self.policyNet

,labels=self.porbPlaceHolder)

self.lossP = lossP = tf.reduce\_mean(cross)

**Annotation:**

Softmax\_cross\_entropy\_with\_logits() is the function to calculate the cross entropy with the net outputs and labels . After complete the calculation , then it will do an softmax function to impart the condition probability of each class .

Then , we will do an mean operation to give the final numeric cost of those classes .

**Example2:**

lossV = tf.reduce\_mean(tf.squared\_difference(self.valueNet,self.valuePlaceHolder))

**Annotation:**

If the final output of your net is a number , and the label is also a numeric , we can use the function which name is squared\_difference , this can calculate the difference between double numbers or the distinguish between the lists which have the same size and the dimension is 1 \* N .After doing that calculate , we also need to do an operation of meaning to output the final cost.

**OPTIMIZER FUNCTION**

**1、Example1:**

opt = tf.train.AdamOptimizer(learning\_rate=0.0001).minimize(lossT)

**2、Example2:**

update\_ops = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

with tf.control\_dependencies(update\_ops):

Opt=tf.train.MomentumOptimizer(momentum=0.9,learning\_rate=self.learningRate,use\_nesterov=True).minimize(lossT)

**Annotation:**

Usually , only use the example1 is fine . However , if there are some layers of BN , we must use the form in the example2 , because there are some others operations or parameters in the layers of BN . if its run with optimizer parallel , it will makes mistake in the calculation of gradient . So , we must optimize its separately .

tf.control\_dependencies(control\_inputs)

This function is used to control the operations in the tensorflow graph . The operation in the control\_inputs will be run first . The operations under it will be run next .

**OTHERS**

**1、Xavier Initialization**

The Xavier initialization is based on the work of Xavier Glorot and Yoshua Bengio in their paper “Understanding the difficulty of training deep feedforward neural networks.” An explanation can be found here. Weights should be initialized in a way that promotes “learning”. The wrong weight initialization will make gradients too large or too small, and make it difficult to update the weights. Small weights lead to small activations, and large weights lead to large ones. Xavier weight initialization considers the distribution of output activations with regard to input activations. Its purpose is to maintain same distribution of activations, so they aren’t too small (mean zero but with small variance) or too large (mean zero but with large variance). DL4J’s implementation of Xavier weight initialization aligns with the Glorot Bengio paper, Nd4j.randn(order, shape).muli(FastMath.sqrt(2.0 / (fanIn + fanOut))). Where fanIn(k) would be the number of units sending input to k, and fanOut(k) would be the number of units recieving output from k.

**2、Fast Fourier Transformation**

**Example:**

import matplotlib.pyplot as plt

Import numpy as np

timeStepSequience = [math.sin(2\*math.pi\*5\*t\*0.01)

+5\*math.sin(2\*math.pi\*10\*t\*0.01)

+3\*math.cos(2\*math.pi\*22\*t\*0.01)

for t in range(100)]

frequentStepSequience = np.fft.fft(timeStepSequience)

frequentStepSequienceABS = abs(frequentStepSequience)

plt.plot(timeStepSequience)

plt.show()

plt.close()

plt.plot(frequentStepSequienceABS)

plt.show()

plt.close()

timeR = np.fft.ifft(frequentStepSequience)

print(timeR)

timeS = []

for time in timeR :

if time >= 0.0 + 0.0j:

timeS.append(abs(time))

else:

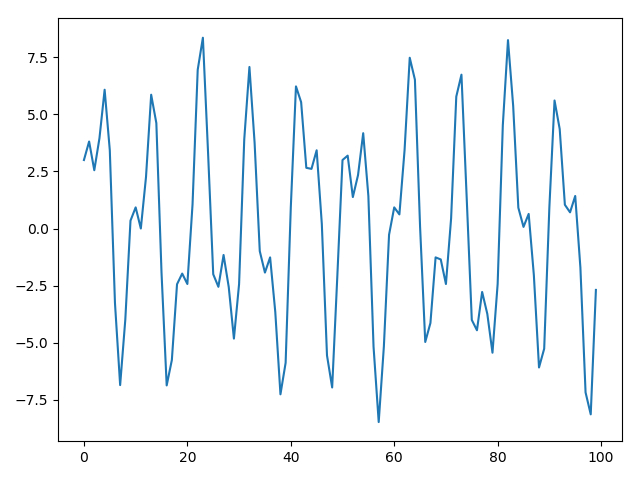
timeS.append(-abs(time))

plt.plot(timeS)

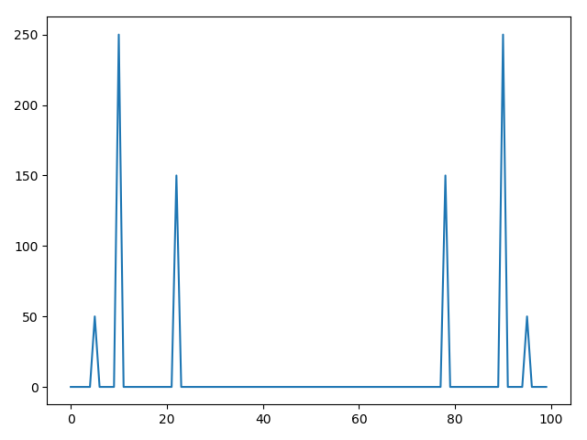
plt.show()

plt.close()

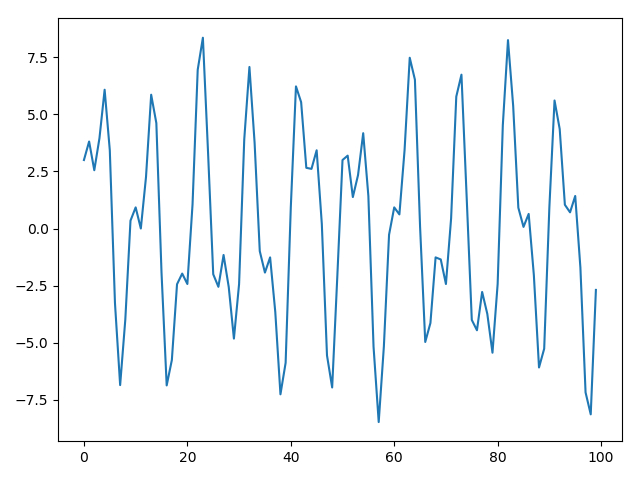
**Result:**

****

**Original Function Plot**

****

**FFT Transform Plot**

****

**IFFT Transform Plot**

1. **Do FFT2 in tensorflow and test if it can do loss operation if use FFT2 in the network . (yes , it can calculate the gradient of the parameters in the net that has used the FFT2D function . )**

**Example:**

var=tf.Variable(initial\_value=tf.truncated\_normal(shape=[1,2,19,19],dtype=tf.float32),trainable=True)

varComplex = tf.complex(real=var,imag=tf.zeros(shape=var.shape))

varFT = tf.fft2d(varComplex)

#simulate the operation of convolution

varFT = varFT \* 0.35

#reverse ifft

varFTR = tf.ifft2d(varFT)

#do abs

varFTABS = tf.abs(varFTR)

varOne = tf.reshape(tensor=varFTABS,shape=[2\*19\*19])

loss = tf.reduce\_mean(varOne)

opt = tf.train.AdamOptimizer(learning\_rate=0.01).minimize(loss)

with tf.Session() as sess:

sess.run(tf.initialize\_all\_variables())

for i in range(5):

print(sess.run(loss))

updataOp = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

with tf.control\_dependencies(updataOp):

sess.run(opt)

print(sess.run(loss))

**Annotation:**

fft2d() will do FFT in the dimensions of the most double inner dimension.if the shape is [1,3,19,19] , the FFT will be used in the dimension of [19,19].

**ONE OF RESULT IS :**

“”

0.26626998

0.26279598

0.26279598

0.25934568

0.25934568

0.25591844

0.25591844

0.2525447

0.2525447

0.24917445

“”

1. **Test number image with fft and do fft\_shift**

**Example:**

image = np.array(line,dtype=np.float32)

image = np.reshape(image, newshape=[28, 28])

imageFFT2 = np.fft.fft2(a=image)

#do shift of fft

imageFFTSh = np.fft.fftshift(imageFFT2)

#for getting a wonderful image , i do an operation of abs ,

#but , if in the real network, it does not need to do this.

#bacause the complex can be multiplied with real number .

imageFFT2abs = abs(imageFFTSh)

plt.imshow(image)

plt.show()

plt.close()

plt.imshow(imageFFT2abs)

plt.show()

plt.close()

#Doing the converse fft operation can reverse the frequent function

#to distribution function -- the original image with abs operation.

#Must use the original fft result with complex ,it can not use the fft complex which #has through the operation of abs to tansform to real number .

reimage = np.fft.ifft2(imageFFTSh)

reimage = abs(reimage)

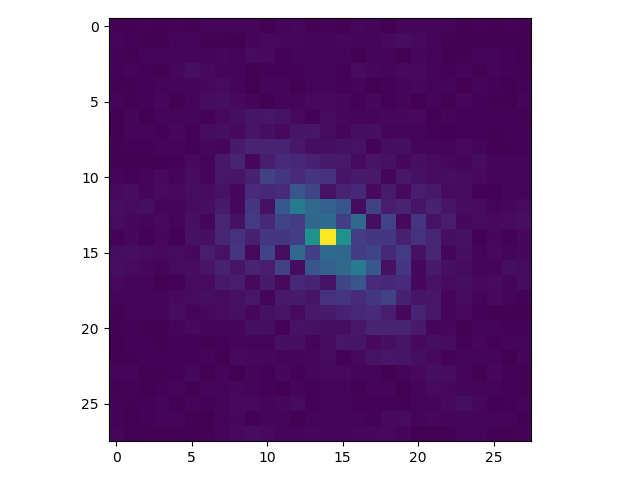
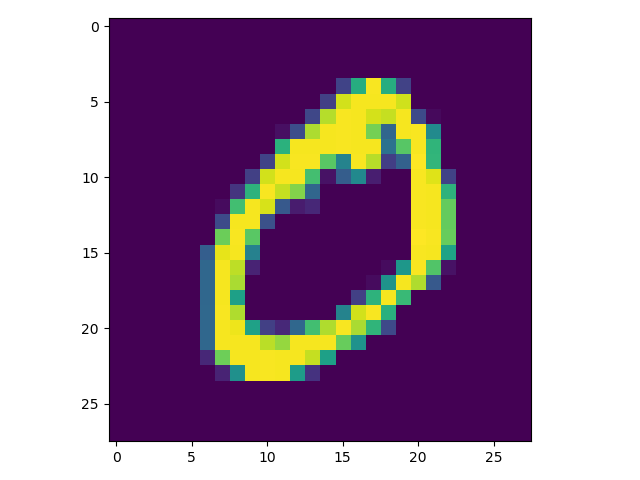
#the image from reversing fft\_shift operation is as same as original image .

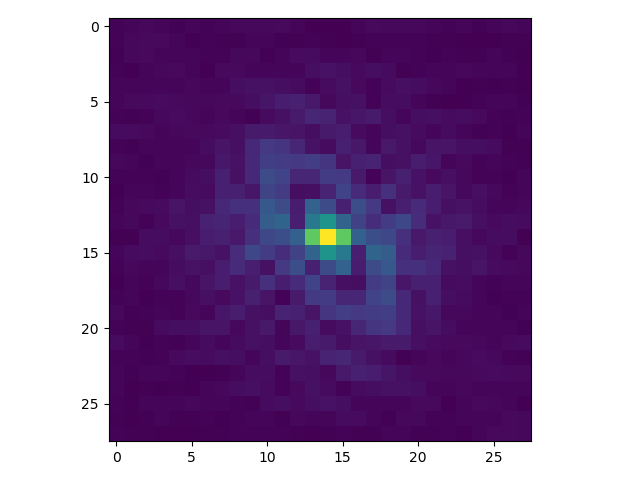
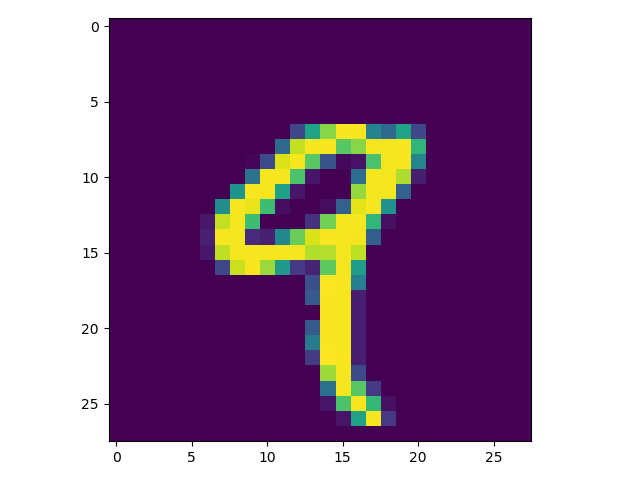
plt.imshow(reimage)

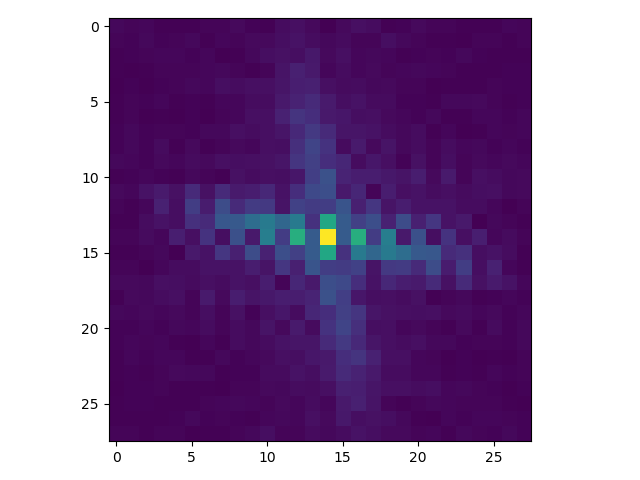
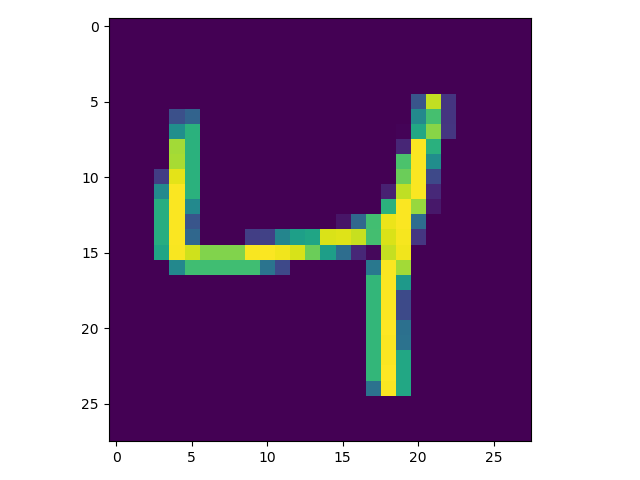
plt.show()

plt.close()

**Result:**







1. **Read data with queue of tensorflow .(read the binary data，too fast )**

**Example :**

#create an tensor to reserve the path of file

varName = tf.Variable(

initial\_value=tf.constant(value="C:\\Users\\Administrator\\Documents\\CIFAR\\cifar-10-binary\\data\_batch\_1.bin", shape=[1], dtype=tf.string))

#create an string input producer

nameIn = tf.train.string\_input\_producer(varName)

#create an fix length reader , it will reade data from input producer

reader = tf.FixedLengthRecordReader(record\_bytes=1 + 32 \* 32 \* 3)

#reade operation , return key and record string . data is in the record string.

key, recordString = reader.read(nameIn)

#decode the record string to unit8

recordBytes = tf.decode\_raw(recordString, tf.uint8)

# cast function is used to transform the type from one type to another .

#slice function is used to cut the data,the second parameter is the begin of position,

#the third parameter is to control the size of how many numbers would you want to cut.

image\_label = tf.cast(tf.slice(recordBytes, [0], [1]), tf.int32)

image\_line = tf.cast(tf.slice(recordBytes, [1], [32 \* 32 \* 3]), tf.int32)

image\_real = tf.reshape(image\_line, shape=[3, 32, 32])

with tf.Session() as sess:

sess.run(tf.initialize\_all\_variables())

#must use this function,if not , the program will not

#start to read the data .

tf.train.start\_queue\_runners(sess=sess)

for i in range(10):

image\_label1 = sess.run(image\_label)

image\_real1 = sess.run(image\_real)

print(image\_label1)

print(image\_real1)